Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Informática e Estatística
INE5416 – Paradigmas de Programação
Graduando 1: Cainã Correa Caldas
Graduando 2: Vinicius Guedes dos Santos

Prova 2 - Lisp

A Solução:

A solução que encontramos foi utilizar um backtracking simples, onde as posições do tabuleiro são preenchidas da esquerda para a direita e de cima para baixo com números em ordem crescente, quando um número é colocado em uma posição, testa-se se alguma regra é quebrada, caso isso aconteça, o programa tenta outro número, ou volta e altera a posição anterior. Caso nenhuma regra seja quebrada, o número testado é mantido na posição e o algoritmo avança para a próxima posição. Para representar o tabuleiro, inspiramo-nos no exemplo visto nas atividades que continha um exemplo de manipulação de matriz com a função ‘maior’. Além das posições vazias, a matriz usada também contém as informações a respeito de quantos prédios podem ser vistos de certa posição. Durante a execução do backtracking, o programa utiliza três diferentes estruturas de dados. Uma delas é o tabuleiro propriamente dito, essa matriz será modificada durante o processo e seu estado final é a solução do problema, a outra estrutura utilizada é uma matriz que mantém informações a respeito de que números já foram testados em cada posição, e por fim, uma pequena lista que contém todos os números possíveis de um determinado cenário. Alguns trechos de código importantes:

```
(defun resolve (k x y m v p s)
  ;(printMatriz m tam) ; remova o ';' para acompanhar o puzzle sendo resolvido
  (setq posConstante (= c (getxym x y v)))
  (setq m-anterior (getxym x y m))
  (setq v-anterior (getxym x y v))
```

```
 (progn
        (setXY (getI (getxym x y v) p) x y m)
        (setXY (+ (getxym x y v) 1) x y v)
        (cond
        ((<= k 0) m) ;escolher m ou v para retornar (chegou no limite da recursão)
        ((and posConstante (= s indo))
        (progn
        (setXY m-anterior x y m)
        (setXY v-anterior x y v)
        (resolve (- k 1) (nextX x) (nextY x y) m v p indo)
        )
        )
        ( (and posConstante (= s voltando))
        (progn
        (setXY m-anterior x y m)
        (setXY v-anterior x y v)
        (resolve (- k 1) (backX x) (backY x y) m v p voltando)
        )
        )
        ((not (getxym x y m))
        (progn
        (setXY o x y m)
        (setXY 0 x y v)
        (resolve (- k 1) (backX x) (backY x y) m v p voltando)
        )
        )
        ((and (tahOk x y m) (= y (- tam 2)) (= x (- tam 2))) m) ; resolveu
        ((not (tahOk x y m)) (resolve (- k 1) x y m v p s))
        ((tahOk x y m) (resolve (- k 1) (nextX x) (nextY x y) m v p s))
        )
 )
)


; tahOk varifica se aquela posição será considerada como certa por enquanto, antes de preencher as próximas
(defun tahOk (x y m)
 (progn
        (setq num (getxym x y m))
    (setq vc (vejoCerto x y m))
        (setXY o x y m)
    (setq ntl (not (jaTemNaLinha num x y m)))
        (setq ntc (not (jaTemNaColuna num x y m)))
        (setq ntd (not (jaTemNasDiagonais num x y m)))
        (setXY num x y m)
        (setq lc (or (< x (- tam 2))  (linhaCerta y m tam)) )
        (setq cc (or (< y (- tam 2)) (colunaCerta x m tam)) )
        (and vc ntl ntc (or ntd (/= (getxym 0 0 m) di)) lc cc)
 )
)
)
```

A função tahOk verifica se a matriz no estado atual está quebrando alguma regra, esta informação é utilizada pela função resolve para saber se é adequado manter um número naquela posição ou se deve-se tentar outro. Esta função chama outras que verificam a corretude das linhas, colunas e possivelmente diagonais.

Como informar a entrada

Existem vários exemplos de entradas no início do código, e de saídas na função main, mas para oficializar, ou em caso de dúvida, é pertinente fazer a leitura:

Para informar a entrada, é preciso alterar o código fonte, adicionando o problema na forma de array (m). Ele deverá conter, nos espaços em branco “-1”, onde o cenário será resolvido. Já nos números das laterais, que indicam quantos prédios devem ser vistos de uma certa posição, onde não houver número nenhum, deve-se colocar o valor da constante “e”. Além disso, para indicar que a solução de um determinado cenário deve levar em consideração as diagonais para avaliar repetiçẽos de número, deve-se preencher a primeira posição da lista que representa o cenário com a constante di, caso contrário, qualquer valor nesta posição será considerado como um cenário em que não se considera as diagonais. Para indicar quais valores podem ser utilizados em um cenário, deve-se criar uma outra lista contendo tais valores (p). É necessário fazer o input de uma matriz (v) que contenha ‘-1’ em todos os pares (x,y) exceto nos que estão nas bordas. Essa matriz visa guardar os índices na lista (p) de números possíveis.

A solução do cenário será impressa na tela após a execução, mas além disso, o código tem alguns exemplos sobre como comparar a solução encontrada com aquela esperada e imprimir na tela se são iguais, isso facilita o teste da solução de várias matrizes de uma vez só.

Exemplo de saída que imprime se a matriz foi resolvida corretamente:

```
(setq tam 4)
(imprima (testa m2 r2 v2 `(1 2) T " m2."))
```

Exemplo de saída que imprime a resolução em si:

```
(setq tam 7)
(printMatriz (resolve limite 1 1 cm233 cv233 cp233 T) tam)
```

Dificuldades encontradas:

Algumas das dificuldades encontradas foram na depuração, pois no trabalho 1, íamos corrigindo as coisas enquanto o programa crescia, e na prova 2, escrevemos o backtracking já com todas as funcionalidades (diagonais verificáveis e números fixos), e isso tornou o código mais incerto. O compilador fornecia pouca informação sobre os erros de execução também. O clisp apresentou um stack overflow que não era um erro do código, pois quando compilado, o programa executou até encontrar a solução.

Comparação com Haskell:

Nós preferimos a linguagem Lisp, porque ela permite criar variáveis, o que é um recurso fundamental, que torna o código absurdamente mais legível. É maravilhoso também poder imprimir no meio da execução de uma função, depois continuar sua execução criando variáveis e retornando então só a última linha. Também parece ser bem mais otimizado alterar uma matriz fixa em relação a estar sempre criando novas matrizes e as passando por parâmetro para as chamadas recursivas do backtracking.